THE UNIVERSITY OF CHICAGO

Lecture 2

Feb 9, 2012.

CHICAGO

PLAN

- 2.1 Newton's method and implications.
- 2.2 Computing Derivatives.
- 2.3 Optimization Code Encapsulation.
- 2.4 Linear Algebra.
- 2.5 Sparse Linear Algebra

2.1 Intro to Methods for Continuous Optimization: Newton' Method

- Focus on continuous numerical optimization methods
  - Virtually ALL of them use the Newton Method idea

Newton's Method

- Idea in 1D:
  - Fit parabola through 3 points, find minimum
  - Compute derivatives as well as positions, fit cubic
  - Use *second* derivatives: Newton by means of Taylor expansion at the current point.

Newton's Method

Interpolating Poly (Taylor)

- At each step:

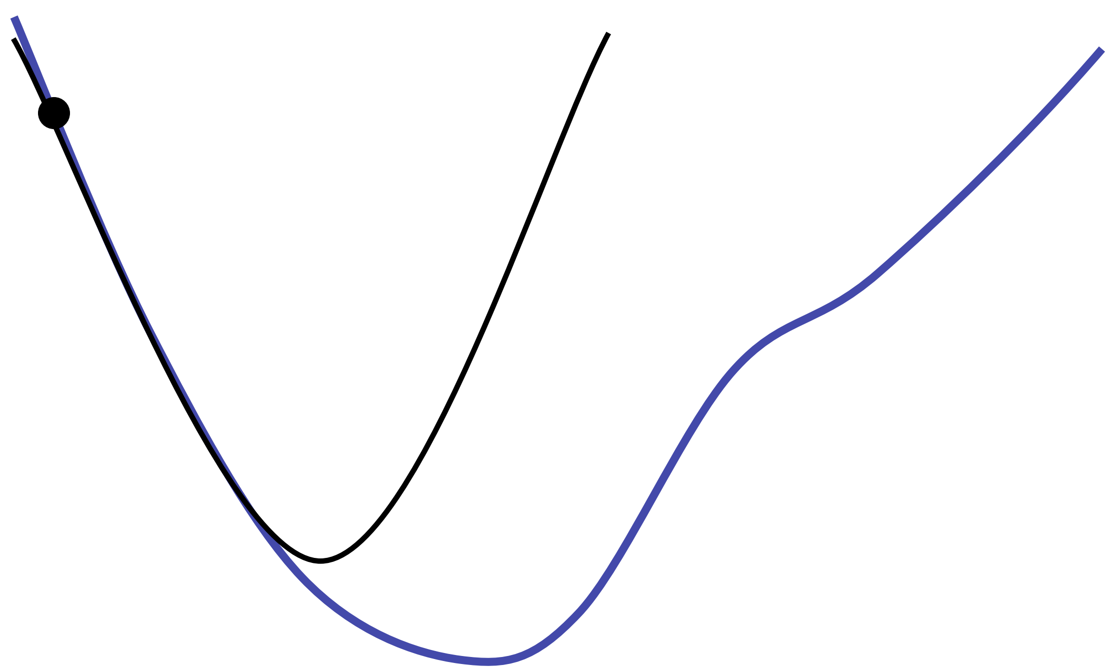
$$\min_x \left[\frac{1}{2}(x - x_k)^2 f''(x_k) + f'(x_k)(x - x_k) + f(x_k) \right]$$

$$\Rightarrow x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

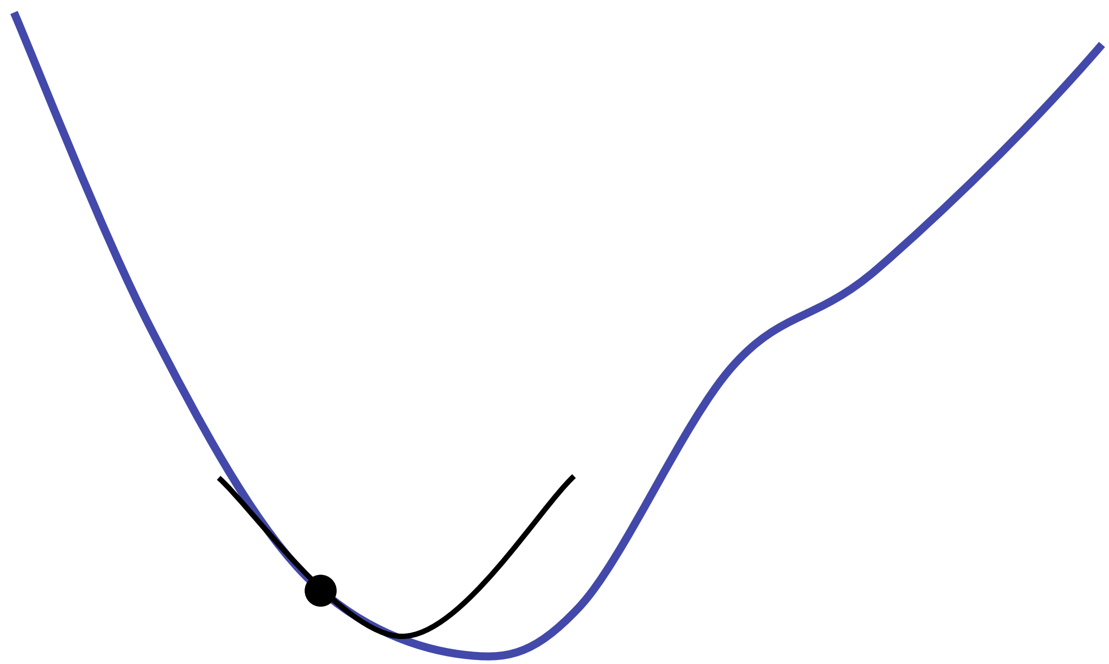
- Requires 1st and 2nd derivatives

CHICAGO

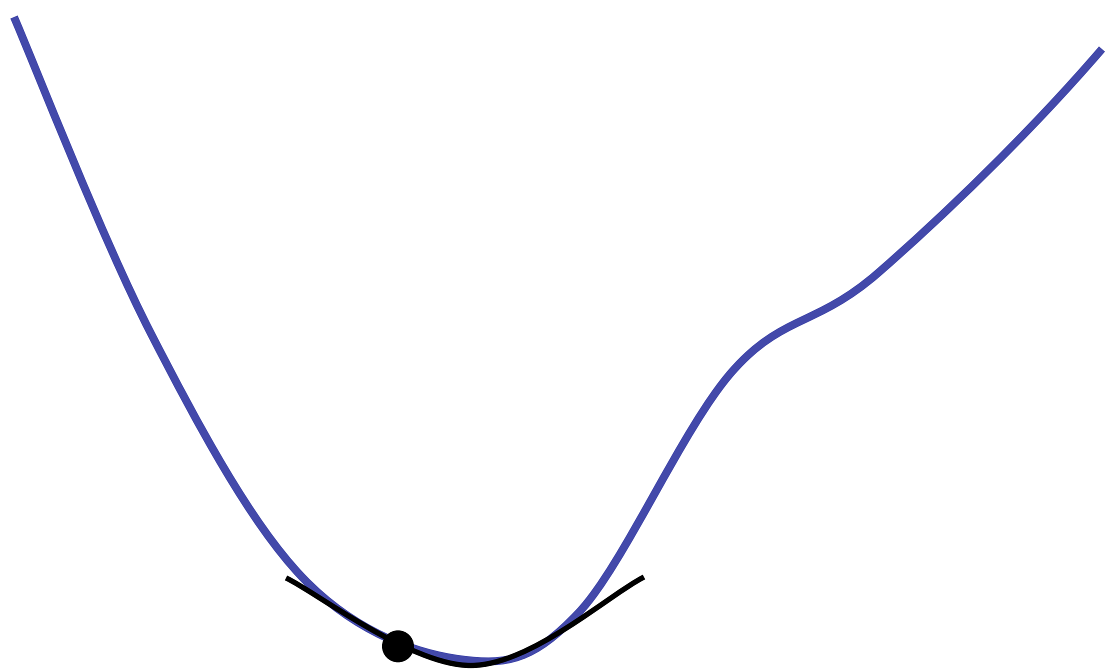
Newton's Method



Newton's Method

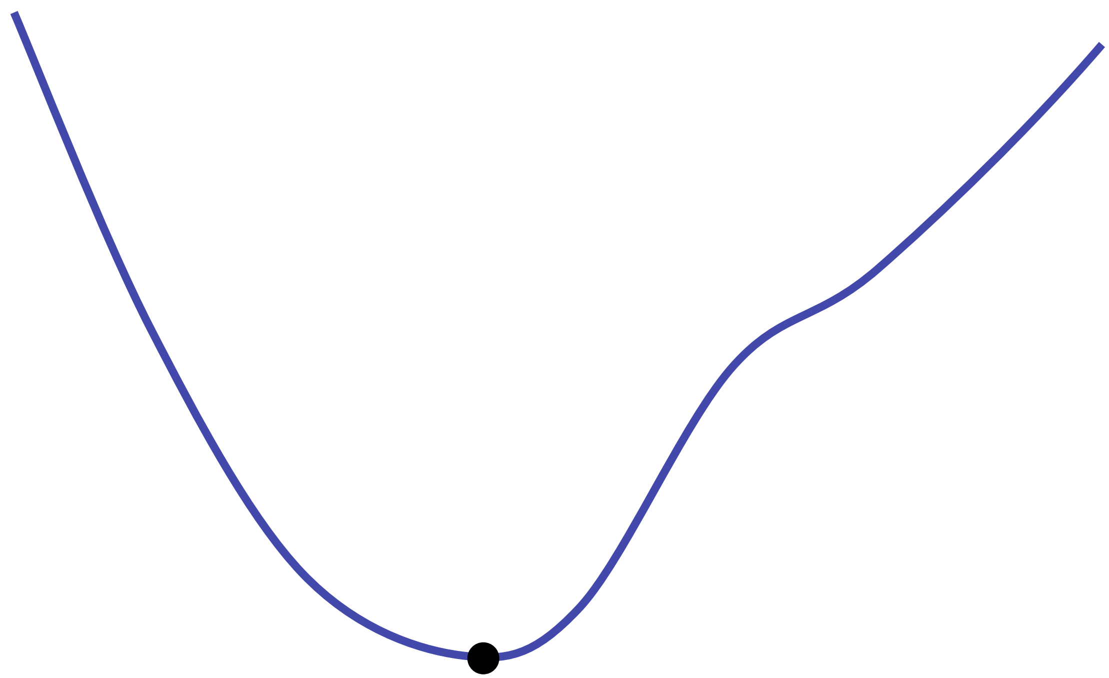


Newton's Method



CHICAGO

Newton's Method



Newton's Method in Multiple Dimensions

- Replace 1st derivative with gradient, 2nd derivative with Hessian

$$f(x, y)$$

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

CHICAGO

Newton's Method in Multiple Dimensions

- Replace 1st derivative with gradient, 2nd derivative with Hessian
- So,

$$\vec{x}_{k+1} = \vec{x}_k - H^{-1}(\vec{x}_k)\nabla f(\vec{x}_k)$$

RECAP: Taylor Series

- The *Taylor series* is a representation of a function as an infinite sum of terms calculated from the values of its derivatives at a single point. It may be regarded as the limit of the Taylor polynomials

Taylor series for a polynominal function, the wt. sum of its derivatives

$$f(x) = \frac{f(x_0)}{0!}(x-x_0)^0 + \frac{f'(x_0)}{1!}(x-x_0)^1 + \frac{f''(x_0)}{2!}(x-x_0)^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n$$

Taylor series for an arbitrary function, any function $\approx$ by the wt. sum of its derivatives

$$f(x) - \frac{f(x_0)}{0!}(x-x_0)^0 = \frac{f'(x_0)}{1!}(x-x_0)^1 + \frac{f''(x_0)}{2!}(x-x_0)^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + R$$

Where $R = \dfrac{f^{(n+1)}(p)}{(n+1)!}(x-x_0)^{n+1}$

CHICAGO

Recap: Multi-dimensional Taylor expansion

A function may be approximated locally by its Taylor series expansion about a point **x***

$$\boxed{f(\mathbf{x}^* + \mathbf{x}) \approx f(\mathbf{x}^*) + \nabla f^T \mathbf{x} + \frac{1}{2}\mathbf{x}^T \mathbf{H} \mathbf{x}}$$

where the gradient $\nabla f(\mathbf{x}^*)$ is the vector

$$\nabla f(\mathbf{x}^*) = \left[\frac{\partial f}{x_1} \dots \frac{\partial f}{x_N}\right]^T$$

and the Hessian **H(x*)** is the symmetric matrix

$$\mathbf{H}(\mathbf{x}^*) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_N} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_N \partial x_1} & \dots & \frac{\partial^2 f}{\partial x_N^2} \end{bmatrix}$$

Q: What is a residual bound? How would you prove it from 1D?

Recap: Orders of convergence

- R-convergence and Q-convergence.
- EXPAND

- Q: Which order of convergence is desirable? Why?

Newton's Method in Multiple Dimensions

- EXPAND: Justify by Quadratic Approximation, and sketch quadratic convergence.
- Tends to be extremely fragile unless function very smooth and starting close to minimum.
- Nevertheless, this iteration is the basis of most modern numerical optimization.

Newton Method: Abstraction and Extension

- "Minimizing a quadratic model iteratively"
- EXPAND
- We need:
  - 1. Derivatives
  - 2. Linear Algebra (to solve for direction).

CHICAGO

NM Implementations

- Descent Methods, Secant Methods may be seen as "Newton-Like"
- All "Newton-like" methods need to solve a linear system of equations.
- All "Newton-like" methods need the implementation of derivative information (unless a modeling language provides it for free, such as AMPL). .

2.2 Computing Derivatives

- Three important ways.
- 1. Hand Coding (rarely done and error prone). Typical failure: do the physics, ignore the design till it is too late.
- 2. Divided differences.
- 3. Automatic Differentiation.

CHICAGO

2.2.1. Divided Differences

The formulas developed next can be used to estimate the value of a derivative at a particular value in the domain of a function, they are primarily used in the solution of differential equations in what called **finite difference methods**.

Note: There a several ways to generate the following formulas that approximate f '(x). The text uses interpolation. Here we use Taylor expansions.

A **difference quotient** is a change in function values divided by the corresponding domain values. For example

$$\frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}.$$

For $y = f(x)$ with $x = x_0$ and x_1 we have

$$\frac{\Delta y}{\Delta x} = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

or for $y = f(x)$ with $x = x_0$ and $x_1 = x_0 + h$ we have

$$\frac{\Delta y}{\Delta x} = \frac{f(x_0 + h) - f(x_0)}{x_0 + h - x_0} = \frac{f(x_0 + h) - f(x_0)}{h}.$$

Note that the last formula also applies in multiple dimensions, if I perturb one coordinate at the time. EXPAND

CHICAGO

Forward Difference Approximation

Given $\mathbf{y = f(x)}$ and $\mathbf{y_h} = \dfrac{\mathbf{f(x_0 + h) - f(x_0)}}{\mathbf{h}}$ for $h > 0$ and some fixed value $\mathbf{x_0}$. Assume also that $|\mathbf{f''(x)}|$ is bounded by a constant C. Show that $\mathbf{f'(x_0) = y_h + O(h)}$. Here we use Taylor's Theorem.

Proof: Expand $\mathbf{f(x_0 + h)}$ using Taylor's Theorem with center of expansion $\mathbf{x_0}$ we get

$$\mathbf{f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(\xi)}$$ where ξ is between x_0 and $x_0 + h$.

Subtract $f(x_0)$ from both sides & divide by h.

It follows then that $\mathbf{y_h} = \dfrac{\cancel{\mathbf{f(x_0)}} + \mathbf{hf'(x_0)} + \frac{\mathbf{h^2}}{2}\mathbf{f''(\xi)} - \cancel{\mathbf{f(x_0)}}}{\mathbf{h}} = \mathbf{f'(x_0) + \frac{h}{2} f''(\xi)}$.

So $\mathbf{y_h = f'(x_0) + \frac{h}{2} f''(\xi)}$. Using that $|\mathbf{f''(x)}| \le \mathbf{C}$ we get that $|\mathbf{f'(x_0) - y_h}| \le \frac{\mathbf{h}}{2}\mathbf{C}$. It then follows that $\mathbf{f'(x_0) = y_h + O(h)}$.

$\boxed{\dfrac{\mathbf{f(x_0 + h) - f(x_0)}}{\mathbf{h}}}$ is called the **Forward Difference Approximation** to $\mathbf{f'(x)}$ at $\mathbf{x = x_0}$.

Finite Differences

- Nevertheless, we use forward differences, particularly in multiple dimensions. (Q: How many function evaluations do I need for gradient?)
- Q: How do we choose the parameter h? EXPAND
- DEMO.
- EXPAND Multiple Dimension Procedure.

2.2.2 Automatic Differentiation

- There exists another way, based upon the chain rule, implemented automatically by a "compiler-like" approach.
- Automatic (or Algorithmic) Differentiation (AD) is a technology for automatically augmenting computer programs, including arbitrarily complex simulations, with statements for the computation of derivatives
- In MATLAB, done through package "intval".

CHICAGO

Automatic Differentiation (AD) in a Nutshell

- Technique for computing analytic derivatives of programs (millions of loc)
- Derivatives used in optimization, nonlinear PDEs, sensitivity analysis, inverse problems, etc.

Automatic Differentiation (AD) in a Nutshell

- AD = analytic differentiation of elementary functions + propagation by chain rule
  - Every programming language provides a limited number of elementary mathematical functions
  - Thus, every function computed by a program may be viewed as the composition of these so-called intrinsic functions
  - Derivatives for the intrinsic functions are known and can be combined using the chain rule of differential calculus

CHICAGO

Automatic Differentiation (AD) in a Nutshell

- Associativity of the chain rule leads to many ways of combining partial derivatives, including two main modes: forward and reverse
- Can be implemented using source transformation or operator overloading

Accumulating Derivatives

- Represent function using a directed acyclic graph (DAG)
- Computational graph
  - Vertices are intermediate variables, annotated with function/operator
  - Edges are unweighted
- Linearized computational graph
  - Edge weights are partial derivatives
  - Vertex labels are not needed
- EXPAND: Example 1D case, + reverse.

CHICAGO

A Small Example

... lots of code...

a = cos(x)

b = sin(y)*y*y

f = exp(a*b)

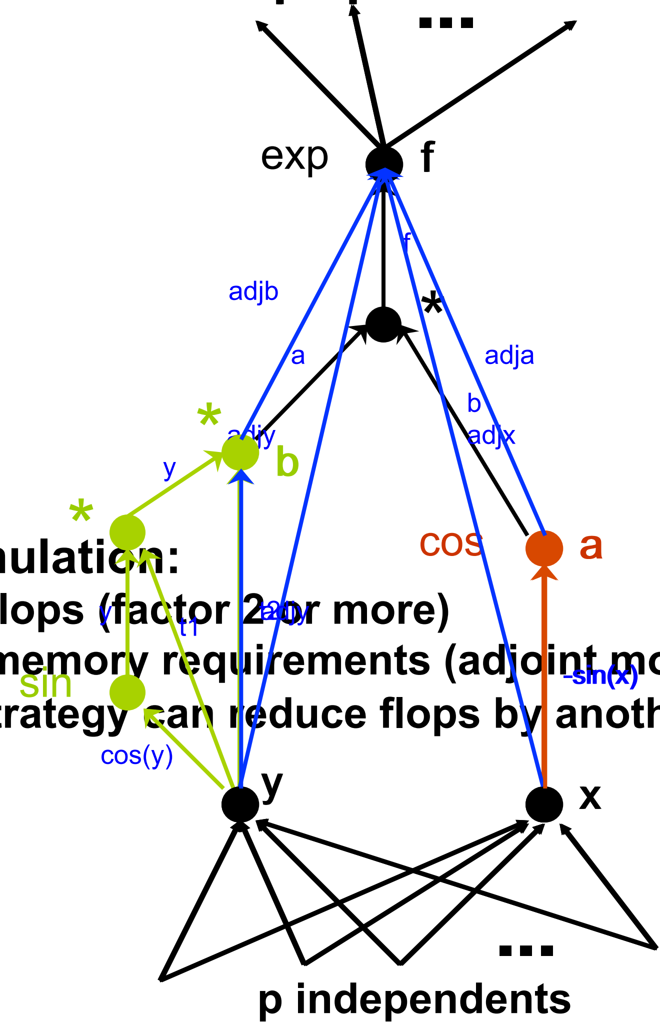
... lots of code...

ADIC: 11 + 17p

New algorithm: 11 + 2p

```
a = cos(x)
  dadx = -sin(x)
  g_a(1:p) = dadx*g_x(1:p)
tmp1 = sin(y)
  d1dy = cos(y)
  g_1(1:p) = d1dy*g_y(1:p)
tmp2 = tmp1*y
  g_2(1:p) = y*g_1(1:p)+tmp1*g_y(1:p)
b = tmp2*y
  g_b(1:p) = y*g_2(1:p)+tmp2*g_y(1:p)
tmp1 = a*b
  g_1(1:p) = b*g_a(1:p)+a*g_b(1:p)
f = exp(tmp1)
  g_f(1:p) = f*g_1(1:p)
```

```
a = cos(x)
  dadx = -sin(x)
tmp1 = sin(y)
  d1dy = cos(y)
tmp2 = tmp1*y
b = tmp2*y
f = exp(a*b)
  adjx = f*b*dadx
  adjy = f*a*(tmp2 + y*(tmp1 + d1dy*y))
  adj2 = y*tmp1 + tmp2
  adja = f*b
  adjb = f*a
  g_f(1:p) = adjx*g_x(1:p)+adjy*g_y(1:p)
  g_f(1:p) = adja*g_a(1:p)+adjb*g_b(1:p)
```

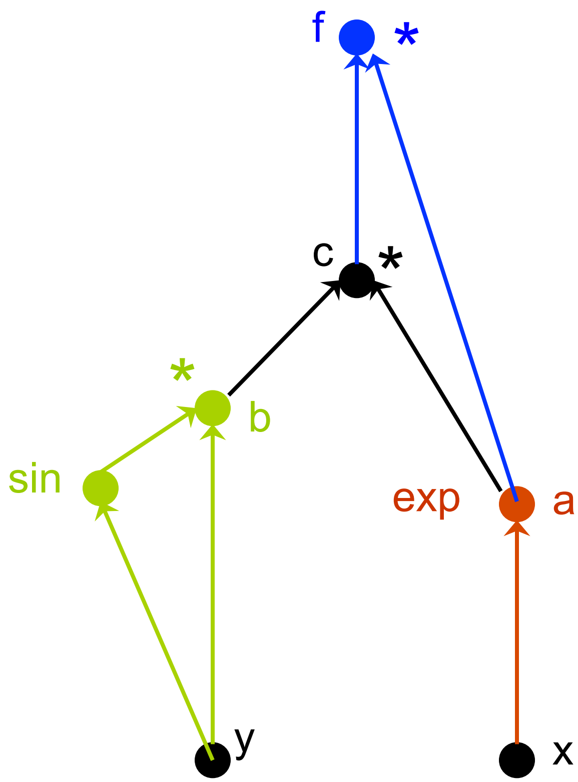


Preaccumulation:

- Reduces flops (factor 2 or more)
- Reduces memory requirements (adjoint mode)
- Optimal strategy can reduce flops by another factor of 2

A simple example

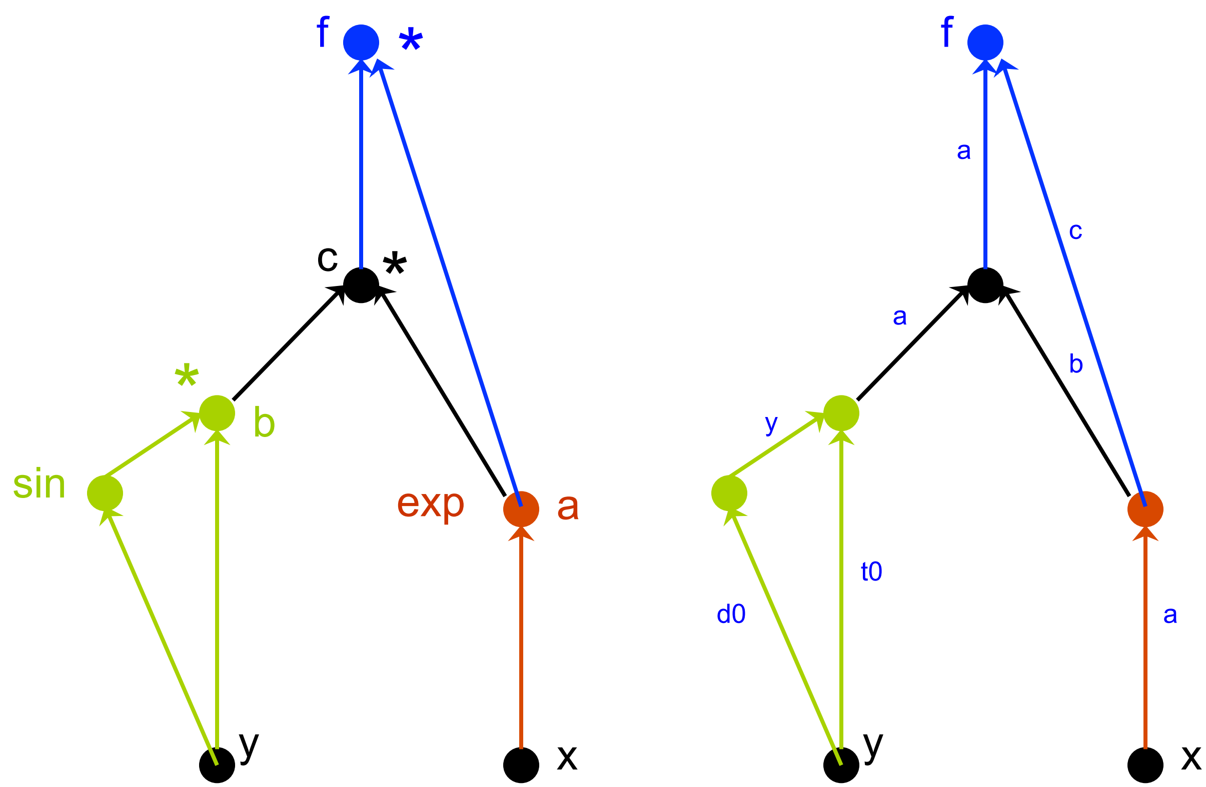
b = sin(y)*y
a = exp(x)
c = a*b
f = a*c



CHICAGO

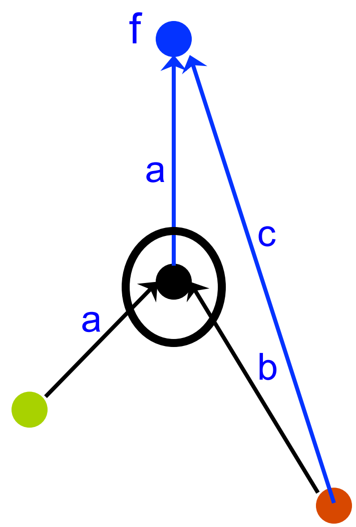
A simple example

```
t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
```



CHICAGO

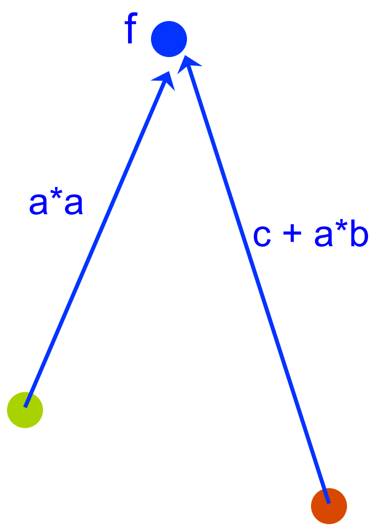
Vertex elimination



- Multiply each in edge by each out edge, add the product to the edge from the predecessor to the successor
- Conserves path weights
- This procedure always terminates
- The terminal form is a bipartite graph

CHICAGO

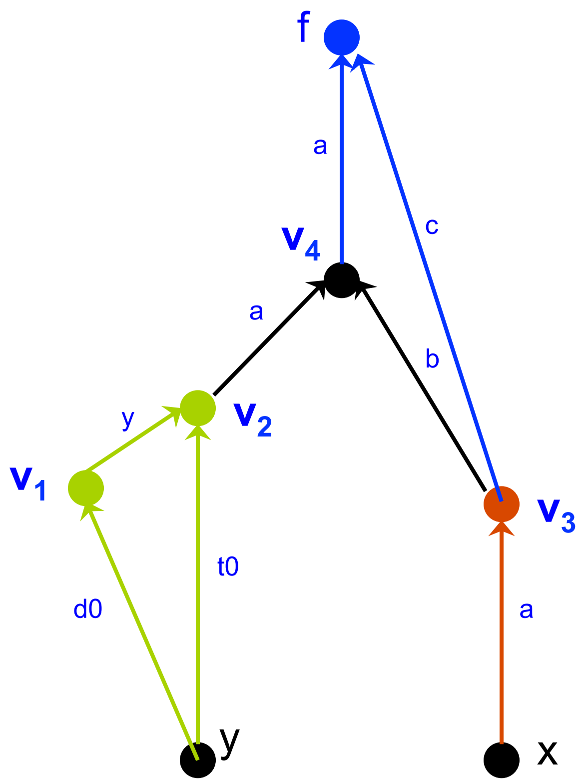
Vertex elimination



- Multiply each in edge by each out edge, add the product to the edge from the predecessor to the successor
- Conserves path weights
- This procedure always terminates
- The terminal form is a bipartite graph

CHICAGO

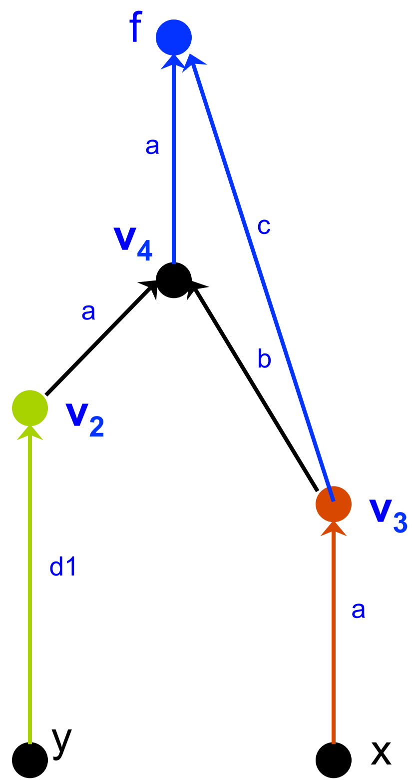
Forward mode: eliminate vertices in topological order



t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c

CHICAGO

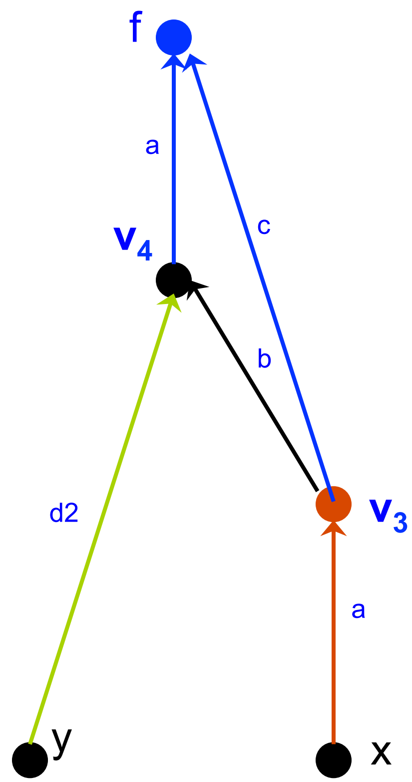
Forward mode: eliminate vertices in topological order



t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = t0 + d0*y

CHICAGO

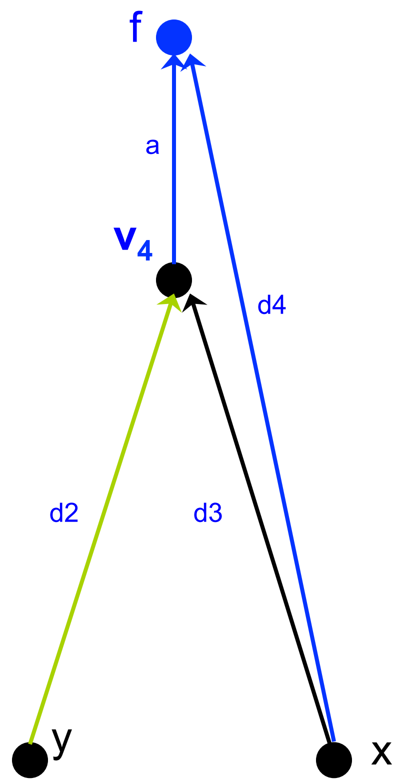
Forward mode: eliminate vertices in topological order



t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = t0 + d0*y
d2 = d1*a

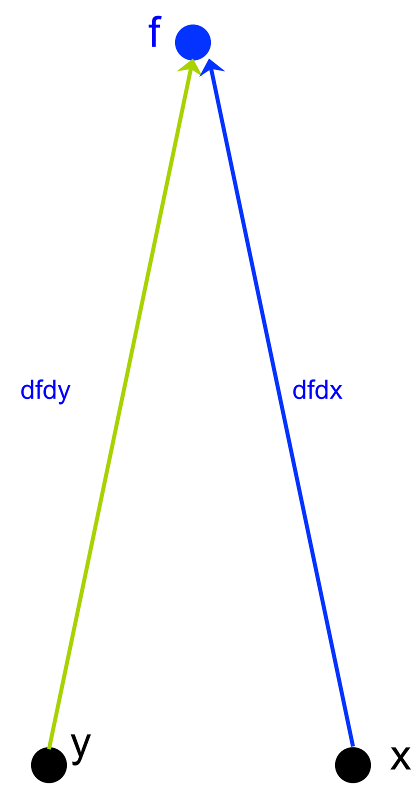
CHICAGO

Forward mode: eliminate vertices in topological order



t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = t0 + d0*y
d2 = d1*a
d3 = a*b
d4 = a*c

Forward mode: eliminate vertices in topological order

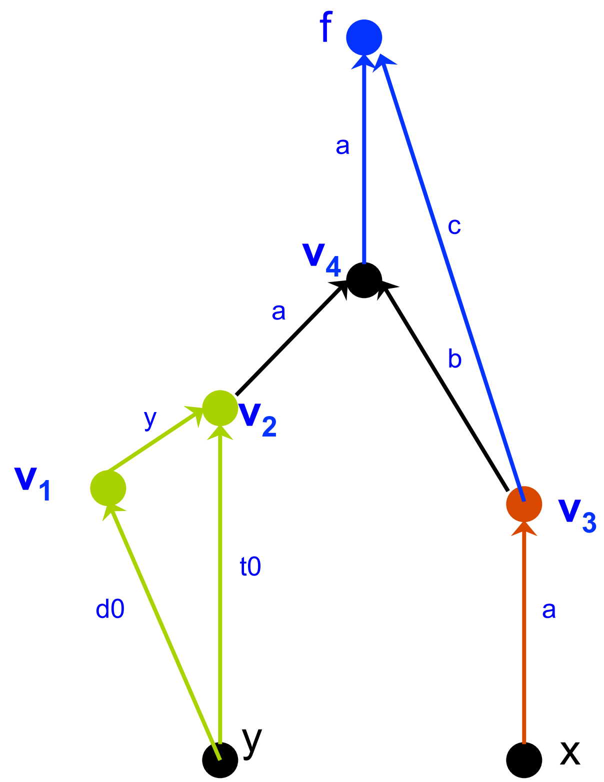


t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = t0 + d0*y
d2 = d1*a
d3 = a*b
d4 = a*c
dfdy = d2*a
dfdx = d4 + d3*a

6 mults 2 adds

CHICAGO

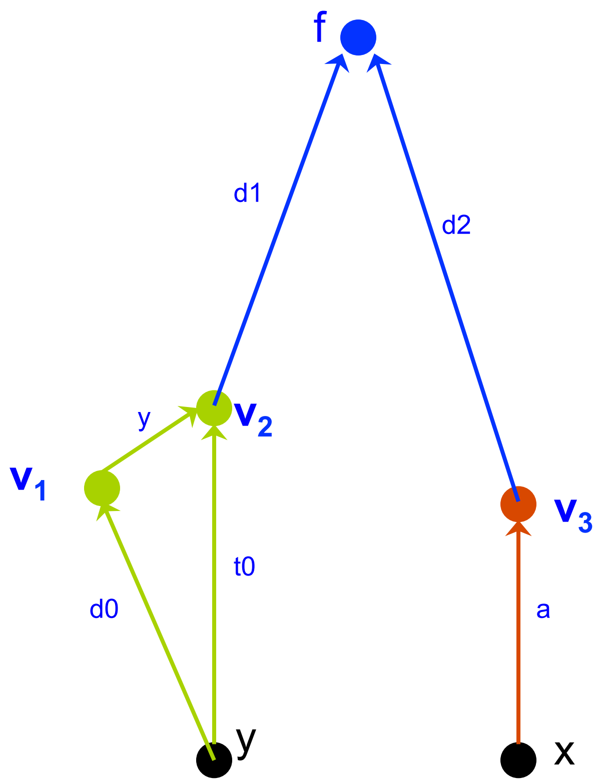
Reverse mode: eliminate in reverse topological order



t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c

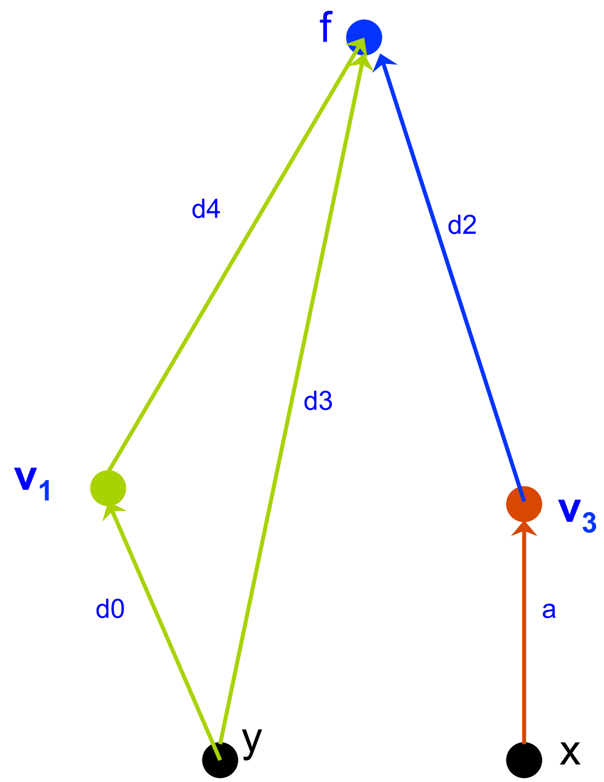
CHICAGO

Reverse mode: eliminate in reverse topological order



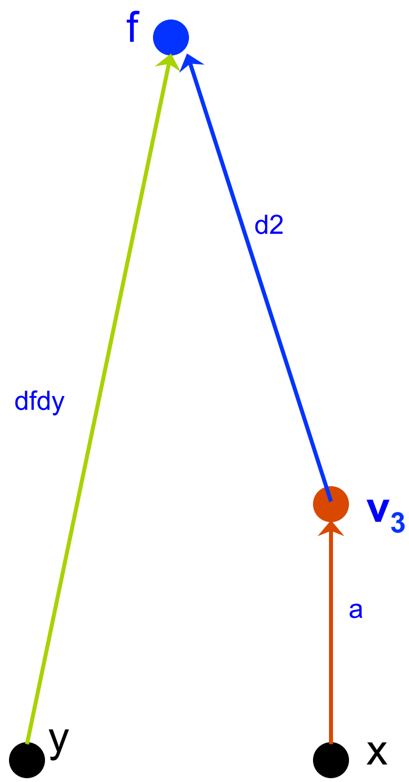
t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = a*a
d2 = c + b*a

Reverse mode: eliminate in reverse topological order



t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = a*a
d2 = c + b*a
d3 = t0*d1
d4 = y*d1

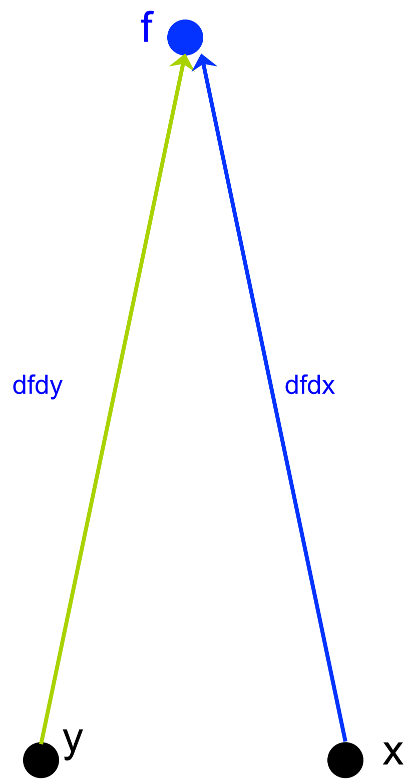
Reverse mode: eliminate in reverse topological order




```
t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = a*a
d2 = c + b*a
d3 = t0*d1
d4 = y*d1
dfdy = d3 + d0*d4
```

CHICAGO

Reverse mode: eliminate in reverse topological order



t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = a*a
d2 = c + b*a
d3 = t0*d1
d4 = y*d1
dfdy = d3 + d0*d4
dfdx = a*d2

6 mults 2 adds

Forward gradient Calculation

- Forward mode computes $\nabla f; \quad f : R^n \to R^m$
  - At a cost proportional to the number of components of f.
  - Ideal when number of independent variables is small
  - Follows control flow of function computation
  - Cost is comparable to finite differences (can be much less, rarely much more)

CHICAGO

Forward versus Reverse

- Reverse mode computes $J = \nabla f; \quad f : R^n \rightarrow R^m$
  - At a cost proportional to m
  - Ideal for $J^T v$, or J when number of dependent variables is small
  - Cost can be substantially less than finite differences
- COST IF m=1 IS NO MORE THAN 5* COST OF FEVAL. EXPAND.

AD versus divided differences

- AD is preferable whenever implementable.
- C, Fortran versions exist.
- In Matlab, free package INTVAL (one of the main reasons not doing C). DEMO
- Nevertheless, sometimes, the source code DOES not exist. (e.g max likelihood).
- Then, divided differences.

Outline

- Homework Questions? Structure of an optimization code (EXPAND)
- Survey
- 2.3 Direct Linear Algebra – Factorization
- 2.4 Sparsity
- 3.1 Failure of vanilla Newton
- 3.2 Line Search Methods
- 3.3 Dealing with Indefinite Matrices
- 3.4 Quasi-Newton Methods

2.3 OPTIMIZATION CODE ENCAPSULATION

Some thoughts about coding

1. Think ahead of time what functionality your code will have, and define the interface properly
2. If portions of code are similar, try to define a function and "refactorize" (e.g the 3 different iterations).
3. Document your code.
4. Do not write long function files; they are impossible to debug (unless very experienced).

CHICAGO

Example Encapsulation

```
function [x,gradNorm]=newtonLikeIteration(functionHandle,xStartPoint,
    iterationType,iterIndex)
% computes one iteration of Newton Like  method with a diagonal
% perturbation
% INPUT:     functionHandle:     (pointer) Function defining problem
%            xStartPoint:        (vector)  The Starting Point
%            iterIndex:          (integer) The index of the iterate
%            iterationType:      k=1:    Newton's Method
%                                k=2:    Hessian peturbed by identity, I
%                                k=3:    Hessian peturbed by o(iterInd)*I
% OUTPUT:    x:                  (vector) Next iteration Point.
```

```
function [xout,iteratesGradNorms]=newtonLikeMethod(functionHandle,
    xStartPoint,iterationType,stopTolerance,maxIterations)
%  [xout,iteratesGradNorms]=newtonLikeMethod(functionHandle,xStartPoint,
    iterationType,stopTolerance,maxIterations)
% PURPOSE:  computes the outcome of a Newton-like Method with a perturbed
    diagonal
% INPUT:     functionHandle:     (pointer) Handle to the optimization
%                                problem to be solved
%            xStartPoint:        (vector) The starting point
%            iterationType:      (integer) The type of the diagonal
    peturbation to be
%                                used
%            stopTolerance:      (scalar) the gradient size at which the
    iteration
%                                will stop.
%            maxIterations:      (integer) The maximum number of iterations
    for which
%                                the algorithm should be run
% OUTPUT:    xout:               (vector) The final output
%            iteratesGradNorms:  (vector) The norms of the gradients
```

[xout,iteratesGradNorms]=newtonLikeMethod(@fenton_wrap,[3 4]',1,1e-12,200)

2.4 SOLVING SYSTEMS OF LINEAR EQUATIONS

CHICAGO

2.3.1 DIRECT METHODS: THE ESSENTIALS

L and U Matrices

- Lower Triangular Matrix

$$[L]=\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{34} & l_{44} \end{bmatrix}$$

- Upper Triangular Matrix

$$[U]=\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{13} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

LU Decomposition for $Ax=b$

- LU **decomposition / factorization**

  $[A]\{x\} = [L][U]\{x\} = \{b\}$
- **Forward substitution**

  $[L]\{d\} = \{b\}$
- **Back substitution**

  $[U]\{x\} = \{d\}$
- **Q:Why might I do this instead of Gaussian elimination?**

Complexity of LU Decomposition

to solve $Ax=b$:

- decompose A into LU -- cost $2n^3/3$ flops
- solve $Ly=b$ for y by forw. substitution -- cost n^2 flops
- solve $Ux=y$ for x by back substitution -- cost n^2 flops

slower alternative:

- compute A^{-1} -- cost $2n^3$ flops
- multiply $x=A^{-1}b$ -- cost $2n^2$ flops

this costs about 3 times as much as LU

CHICAGO

Cholesky LU Factorization

- **If [A] is symmetric and positive definite, it is convenient to use Cholesky decomposition.**

$$[A] = [L][L]^T = [U]^T[U]$$

- **No pivoting or scaling needed if [A] is symmetric and positive definite (all eigenvalues are positive)**
- **If [A] is not positive definite, the procedure may encounter the square root of a negative number**
- **Complexity is ½ that of LU (due to symmetry exploitation)**

CHICAGO

Cholesky LU Factorization

- $[A] = [U]^T[U]$
- **Recurrence relations**

$$u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2}$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj}}{u_{ii}} \quad for \ \ j = i+1, \ldots, n$$

Pivoting in LU Decomposition

- **Still need pivoting in LU decomposition (why?)**
- **Messes up order of [L]**
- **What to do?**
- **Need to pivot both [L] and a permutation matrix [P]**
- **Initialize [P] as identity matrix and pivot when [A] is pivoted. Also pivot [L]**

LU Decomposition with Pivoting

- Permutation matrix [P]

  - permutation of identity matrix [I]
- Permutation matrix performs "bookkeeping" associated with the row exchanges
- Permuted matrix [P] [A]
- LU factorization of the permuted matrix

  $$[P][A] = [L][U]$$
- Solution

  $$[L][U]\{x\} = [P]\{b\}$$

LU-factorization for real symmetric Indefinite matrix A (constrained optimization has saddle points)

$LU-$ factorization $$A=\left(\begin{array}{c|c} E & c^T \\ \hline c & B \end{array}\right)=\left(\begin{array}{c|c} I & \\ \hline cE^{-1} & I \end{array}\right)\left(\begin{array}{c|c} E & c^T \\ \hline & B-cE^{-1}c^T \end{array}\right)$$

LDL^T- factorization $$A=\left(\begin{array}{c|c} E & c^T \\ \hline c & B \end{array}\right)=\left(\begin{array}{c|c} I & \\ \hline cE^{-1} & I \end{array}\right)\left(\begin{array}{c|c} E & \\ \hline & B-cE^{-1}c^T \end{array}\right)\left(\begin{array}{c|c} I & E^{-1}c^T \\ \hline & I \end{array}\right)$$

where $$L=\left(\begin{array}{c|c} I & \\ \hline cE^{-1} & I \end{array}\right)$$ and $$L^T=\left(\begin{array}{c|c} I & E^{-T}c^T \\ \hline & I \end{array}\right)=\left(\begin{array}{c|c} I & E^{-1}c^T \\ \hline & I \end{array}\right)$$

Question: 1) If A is not singular, can I be guaranteed to find a nonsingular principal block E after pivoting? Of what size?

2) Why not *LU*-decomposition?

CHICAGO

History of LDL' decomposition: 1x1, 2x2 pivoting

- diagonal pivoting method with complete pivoting:
  Bunch-Parlett, "Direct methods fro solving symmetric indefinite systems of linear equations," SIAM J. Numer. Anal., v. 8, 1971, pp. 639-655

- diagonal pivoting method with partial pivoting:
  Bunch-Kaufman, "Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems," Mathematics of Computation, volume 31, number 137, January 1977, page 163-179

- DEMOS

2.4 COMPLEXITY OF LINEAR ALGEBRA; SPARSITY

Complexity of LU Decomposition

to solve $Ax=b$:

- decompose A into LU -- cost $2n^3/3$ flops
- solve $Ly=b$ for y by forw. substitution -- cost n^2 flops
- solve $Ux=y$ for x by back substitution -- cost n^2 flops

slower alternative:

- compute A^{-1} -- cost $2n^3$ flops
- multiply $x=A^{-1}b$ -- cost $2n^2$ flops

this costs about 3 times as much as LU

CHICAGO

Complexity of linear algebra

lesson:

- if you see A^{-1} in a formula, read it as "solve a system", not "invert a matrix"

Cholesky factorization -- cost $n^3/3$ flops

LDL' factorization -- cost $n^3/3$ flops

Q: What is the cost of Cramer's rule (roughly)?

Sparse Linear Algebra

- Suppose you are applying matrix-vector multiply and the matrix has lots of zero elements
  - Computation cost? Space requirements?
- General sparse matrix representation concepts
  - Primarily only represent the nonzero data values (nnz)
  - Auxiliary data structures describe placement of nonzeros in "dense matrix"
- And *MAYBE* LU or Cholesky can be done in O(nnz), so not as bad as (O(n^3)); since very oftentimes nnz=O(n)

Sparse Linear Algebra

- Because of its phenomenal computational and storage savings potential, sparse linear algebra is a huge research topic.
- VERY difficult to develop.
- Matlab implements sparse linear algebra based on i,j,s format.
- DEMO
- Conclusion: Maybe I can SCALE well … Solve O(10^12) problems in O(10^12).

SUMMARY SECTION 2

- The heaviest components of numerical software are Numerical differentiation (AD/DIVDIFF) and linear algebra.
- Factorization is always preferable to direct (Gaussian) elimination.
- Keeping track of sparsity in linear algebra can enormously improve performance.